Évaluation de la méthodologie de placement sur GPU

Sommaire

Nous avons détaillé dans le chapitre 3 notre méthodologie de portage d'algorithmes sur accélérateurs de type GPU. Celle-ci permet, à partir d'un algorithme séquentiel, de transformer certains nids de boucles en *kernels*. Pour cela, nous avons défini trois critères de placement portant sur la structure des nids de boucles, la taille de leur domaine d'itérations ainsi que leurs empreintes mémoire. En complément, nous avons défini un ensemble de transformations de code visant à améliorer le nombre de nids de boucles candidats mais aussi la qualité du placement sur GPU. Afin d'alimenter ce processus de placement, plusieurs analyses statiques sont utilisées. Elles permettent de générer une représentation graphique du code facilitant le déroulement de cette méthodologie. Enfin, les analyses dynamiques permettent de vérifier que chaque *kernel* généré engendre bien un gain en terme de temps d'exécution.

Dans ce chapitre, nous évaluons cette méthodologie de placement sur GPU. Dans un premier temps, nous introduisons dans la section 4.1 les architectures utilisées. Nous présentons ensuite dans la section 4.2, les deux applications ayant servi à l'évaluation de notre méthodologie. Enfin, nous détaillons les résultats pour ces deux applications dans les sections respectives 4.3 et 4.4. Nous évaluons en particulier les analyses statique et dynamique, les critères de placement ainsi que les transformations de code.

4.1 Architectures expérimentales utilisées

Nous avons utilisé deux architectures GPU Nvidia pour les expérimentations. Ce choix est justifié, dans le chapitre 1, par la présence forte de Nvidia dans le secteur industriel. Ce concepteur offre des solutions pour une large gamme d'application telles que l'embarqué ou le calcul scientifique. L'évaluation de notre méthodologie compare les résultats obtenus pour une architecture GPU "classique", sur une plateforme de type *workstation* (Endicott), avec ceux d'une architecture GPU basse consommation, sur une plateforme embarquable (Jetson TX1).

Les caractéristiques de la plateforme **Endicott** sont présentées dans la section 4.1.1 et celles de la **Jetson TX1** dans la section 4.1.2.

4.1.1 Endicott

Endicott est une *workstation*, équipée d'un processeur Intel $64\,bits$ de génération *Haswell*. Il s'agit d'un **Core i7 4770s** composé de 4 cœurs physiques et de 8 cœurs logiques grâce à la technologie d'*HyperThreading*. Sa fréquence de fonctionnement nominale est de $3.1\,GHz$ et peut être augmentée grâce à la technologie *Turbo Boost* à $3.9\,GHz$ sur des périodes de temps réduites. Sa puissance de calcul maximale pour des opérations flottantes simple précision est de $\mathbf{499,2\,GFlops}$. Son Thermal Design Power (TDP) est de $\mathbf{65\,W}$. Enfin sa bande passante mémoire théorique est de $\mathbf{25,6\,GB.s^{-1}}$. La puissance de ce processeur hôte permet d'exploiter le GPU de cette plateforme en minimisant le risque de sous-alimentation des *pipelines* de communication entre l'hôte et l'accélérateur GPU.

Cette station de travail est aussi équipée d'un GPU **Quadro K2000** développé par Nvidia. Cette carte est basée sur une architecture Kepler [129] GK107. La K2000 intègre deux unités Next Generation Streaming Multiprocessor (SMX) (visibles dans la figure 4.1) de 192 cœurs portant ainsi le total à 384 cœurs Cuda par processeur. La puissance de calcul maximale pour les opérations flottantes simple précision est de $\mathbf{732,67\,GFlops}$. La K2000 embarque $2\,GB$ de mémoire dédiée dont la bande passante maximale est de $\mathbf{64\,GB.s^{-1}}$ pour un bus mémoire de $128\,bits$. Enfin, le TDP de ce GPU est de **51W**.

4.1.2 Jetson TX1

La Jetson TX1 est une plateforme d'évaluation conçue par Nvidia pour des applications embarquées. Elle intégre un processeur basse consommation Tegra X1 [121] basé sur le SOC T210 de la marque. Ce dernier embarque quatre cœurs **ARM A57** cadencés à $1.9\,GHz$. Afin de réduire la consommation énergétique globale du SOC T210, les cœurs A57 peuvent être désactivés pour utiliser en remplacement quatre cœurs ARM A53 cadencés à $1.3\,GHz$. La sélection des cœurs A57 ou A53 sera déclenchée, à l'initiative exclusive du processeur, en fonction du taux d'occupation de ces ressources. La puissance de calcul maximale est alors de $\mathbf{60,8GFlops}$.

Sur ce même SOC est aussi incorporé un GPU basé sur l'architecture *Maxwell* [130] **GM20B**. La figure 4.2 illustre la configuration de l'un des Maxwell Streaming Multi-

FIGURE 4.1 – Vue d'un cluster SMX de l'architecture Nvidia Kepler de première génération utilisée pour les Quadro K2000

processor (SMM) spécifique à cette architecture. Chacun d'entre-eux est composé de 128 cœurs CUDA dont la fréquence de fonctionnement est de $1GHz$. Deux unités SMM sont présentes dans le processeur de la Tegra X1 pour un total de 256 cœurs CUDA. La puissance de calcul maximale pour la partie GPU est de **512 GFlops** pour des opérations flottantes simple précision.

En complément, une unique **mémoire unifiée** de $4\,GB$ est partagée par l'ensemble des processeurs précedemment cités. Sa bande passante est de $\mathbf{25.6\,GB.s^{-1}}$ pour un bus mémoire de $64\,bits$.

Le **TDP global** du processeur Tegra X1 est de **15 W**.

4.1.3 Comparaison des architectures

Le tableau 4.1 résume les principales caractéristiques pour les deux plateformes précédemment décrites. Celles-ci présentent certaines caractéristiques intéressantes pour nos expérimentations.

Le domaine d'application, notamment, n'est pas le même pour les deux plateformes. La Jetson TX1 répond à des problématiques de consommation énergétique réduite propre à l'embarqué. Endicott, au contraire, est une plateforme de type *workstation* favorisant les performances calculatoires.

Au niveau de la puissance des processeurs hôte, l'ARM A57 du SOC T210 a une

FIGURE 4.2 – Vue d'un cluster SMM de l'architecture Nvidia Maxwel de seconde génération utilisée pour la Tegra X1

puissance de calcul plus faible(60 $GFlops$) que le Core i7 d'Intel(500 $GFlops$). Le pilotage du GPU par l'A57 est donc plus délicat avec un risque de sous alimentation des pipelines de communication hôte/accélérateur.

Concernant les GPUs, l'architecture *Maxwell* de la Tegra X1 est plus récente que l'architecture *Kepler* de la Quadro K2000. Cela se traduit par quelques différences notamment au niveau de la disposition des mémoires cache. Dans le cas de l'architecture *Kepler*, le cache L1 est partagé avec la *shared memory*. Sur l'architecture *Maxwell*, le même cache est communalisé avec le cache de la *texture memory*. Le nombre de canaux d'accès, pour ce dernier cache, est aussi moins élevé sur l'architecture *Maxwell*. Cependant la taille globale pour chaque SMM ne varie pas. Nous évaluons en particulier les conséquences sur les temps d'accès aux différents espaces mémoires dans la section 5.1.

Les deux GPUs intègrent deux unités SM composées chacune de quatre *warp schedulers* pour huit *instruction dispatch units*. Les fréquences de fonctionnement étant similaires et le cache d'instructions restant identique, le débit de placement des *warps* ne devrait ainsi pas évoluer. L'utilisation des *warp schedulers* et des *instruction dispatch units* pour du

	Endicott		Jetson TX1		
	cpu	gpu	cpu	cpu	gpu
Concepteur	Intel	Nvidia	ARM	ARM	Nvidia
Architecture	Haswell	GK107	v8-A	v8-A	GM20B
Catégorie	Core i7	Quadro	Cortex	Cortex	Tegra
Modèle	4770s	K2000	A53	A57	T210
Compute Capability		3.0			5.3
Nb. cœurs	4	384	4	4	256
Nb. threads	8	4096	4	4	4096
Fréquence Nom.(GHz)	3.1	0.954	1.3	1.9	1.0
Fréquence Max.(GHz)	3.9				
Puissance calc. SP Max.(GFlops)	**499.2**	**732.67**	**41.6**	**60.8**	**512**
L1I cache(KB)	32	16	48		16
L1D cache(KB)	32	16/32/48	32		24
L2 cache(MB)	1	0.256	2		0.256
Qté. mémoire(GB)	8	2	4		
Type	DDR3	GDDR5	LPDDR4		
Fréquence(GHz)	0.8	1.0	1.6		
Data Rate	2	4	2		
Bus mémoire(bits)	128	128	64		
Bande passante(GB/s)	**25.6**	**64**	**25.6**		
Gravure(nm)	22	28	20		
TDP max(W)	65	51	15		

TABLE 4.1 – Tableau récapitulatif des architectures expérimentales utilisées.

parallèlisme *coarse grain* est étudié en particulier dans la section 5.2.

Comparativement à l'architecture *Maxwell*, l'architecture *Kepler* embarque 64 *CUDA cores* supplémentaires par unité SM. Nous noterons aussi que le nombre de *texture units* est moins élevé sur l'architecture *Maxwell* ce qui peut se traduire par une bande passante plus faible pour la *texture memory*, lors de calculs d'interpolation ou de réplication de données.

Les performances générales de ces GPUs sont plutôt limitées, quand on les compare, à génération identique, aux plus puissants GPUs tels que les Quadro K6000 et M6000. Les effets d'un mauvais placement sont ainsi amplifiés par une bande passante mémoire et une fréquence de fonctionnement plus faibles.

La Jetson TX1 présente une mémoire homogène unifiée, réduisant les coûts de transfert mémoire. Endicott au contraire présente une mémoire hétérogène. Les transferts mémoire transitent, dans ce dernier cas, par le port Peripheral Component Interconnect express (PCIe). Ce dernier peut constituer un facteur limitatif.

Enfin, pour plus de détails sur l'architecture des deux GPUs, nous avons ajouté les caractéristiques du SMX de la K2000 et du SMM de la TX1, dans le tableau 4.2. La principale différence entre ces deux processeurs porte sur la quantité de registres utilisables et sur les caractéristiques de la *shared memory*.

Pour les deux architectures :

— chaque *warp* est constitué au maximum de 32 *threads* et

— chaque *block* est constitué au maximum de 1024 *threads*.

Ces caractéristiques sont identiques aux paramètres utilisés pour le second critère de pla-

Compute Capability	3.0	5.3
SM Version	sm_30	sm_53
Threads / Warp	32	32
Warps / Multiprocessor	64	64
Threads / Multiprocessor	2048	2048
Thread Blocks / Multiprocessor	16	32
Shared Memory / Multiprocessor (bytes)	49152	65536
Max Shared Memory / Block (bytes)	49152	49152
Register File Size / Multiprocessor (32-bit registers)	65536	65536
Max Registers / Block	65536	32768
Register Allocation Unit Size	256	256
Register Allocation Granularity	warp	warp
Max Registers / Thread	63	255
Shared Memory Allocation Unit Size	256	256
Warp Allocation Granularity	4	4
Max Thread Block Size	1024	1024
Concurrent kernel execution	16	16
Shared Memory Size Configurations (bytes)	49152 (32768) (16384)	65536
Warp register allocation granularities	256	256

TABLE 4.2 – Caractéristiques du SM 3.0 et du SM 5.3 *Source: Nivida [119]*

cement dans la section 3.3.2.

De même, pour le premier critère de placement (section 3.3.2), les 3 dimensions utilisées pour la répartition des *blocks* et des instances de *threads* sont adaptés à ces deux architectures. De ce fait, nous considérons pour ce critère $B = 3$ et $T = 3$.

Enfin, pour le troisième critère de placement, nous considèrons pour les deux architectures $M^{acc/global} = 1.75\,GB$ comme quantité de mémoire allouable. La mémoire globale ($4\,GB$) de la Jetson TX1 étant partagée entre l'hôte et l'accélérateur, nous avons retenu une répartition équitable ($2\,GB/2\,GB$) de celle-ci. La mémoire de la Quadro K2000 est de $2\,GB$. Pour les deux plateformes, nous préservons $250\,MB$ pour la session graphique.

4.2 Applications étudiées

Deux algorithmes ont été utilisés pour l'évaluation de notre méthodologie. L'algorithme de flot optique *simpleflow* présenté en section 4.2.1 et l'algorithme de variance locale présenté en section 4.2.2.

4.2.1 Algorithme de flot optique

Nous avons utilisé pour nos expérimentations l'algorithme du *simpleFlow* de Tao et al.[150]. L'intégralité du code source, ayant servi de référence, est contenu dans l'annexe A. Celui-ci provient du dépot officiel des contributions [2] à la bibliothèque de traitement d'images OpenCV. Cet algorithme effectue, à partir de deux images distinctes de même taille, le calcul du flot optique dense. Le résultat est alors représenté dans une table contenant pour chaque pixel, son déplacement horizontale et verticale entre les deux images en entrée, dans les coordonnées pixeliques de ces dernières.

Cet algorithme a été retenu car il présente de nombreuses caractéristiques intéressantes pour l'évaluation de notre méthodologie :

- Nous sommes étranger au développement de cet algorithme et n'avons de ce fait eu aucune influence sur sa conception.
- Son développement n'a pas été conçu pour du *benchmarking*. C'est un cas concret d'application de traitement d'image en C++. Le C++ est de plus en plus utilisé pour les applications industrielles.
- Ses 600 lignes de codes représentent un algorithme de complexité importante.
- Son découpage en multiples sous-fonctions permet d'étudier le comportement interprocédural de notre méthodologie.
- Les multiples nids de boucles ayant en moyenne 6 niveaux de profondeur, représentent une source variée de *kernels* potentiels.
- Les domaines d'itérations des boucles de l'algorithme ont des tailles variées.
- Le code source présente du contrôle dynamique avec :
  - des bornes de boucles variables et
  - des branchements non prédictibles.
- Son temps d'exécution, pouvant facilement atteindre plusieurs dizaines de secondes, présente un grand intérêt à être accéléré.
- Sa grande quantité de communications mémoire permet de vérifier la bonne prise en compte des temps de transfert hôte/accélérateur dans le cadre de cette méthodologie.

Enfin, les algorithmes de flot optique présentent de manière globale un fort potentiel, en témoigne les nombreuses publications à leur sujet. La grande quantité de données, mais aussi de calculs, à traiter a donné lieu à de nombreuses études, telles que [149, 134] utilisant le GPU ou encore [135, 136], démontrant les capacités du CPU à supporter une telle charge. Une nouvelle approche émerge, à l'heure actuelle, portant sur l'utilisation conjointe, ou en remplacement, de réseaux de neurones. Nous citerons [51, 77] à titre d'exemple et de manière non exhaustive.

On retrouve, couramment, l'utilisation d'algorithmes de flot optique, notamment pour des applications de Simultaneous Localization And Mapping (SLAM), de stabilisation d'images ou encore de détection d'objets mobiles. Ils sont aujourd'hui employés, pour ce type d'applications, dans plusieurs systèmes optroniques embarqués chez Safran ou encore pour la conduite autonome de véhicules chez Tesla.

4.2.2 Algorithme de calcul de variance locale

L'algorithme de calcul de variance locale [64, 63] est une application spatiale de traitement d'images. Son principe est de calculer, pour chaque *pixel* d'une image, la variance de son voisinage. On le retrouve dans de nombreuses applications telles que la détection d'anomalies en statistiques ou encore pour l'amélioration de contrastes en traitement d'images chez Safran.

Contrairement à un algorithme de convolution, considérant classiquement des fenêtres de 3×3 ou 5×5 éléments, la taille du voisinage est ici bien plus importante, en considérant des voisinages de plusieurs centaines d'éléments. De plus, la tendance à l'augmentation quadratique des résolutions de capteurs d'images a un impact direct sur la taille de ce voisinage en traitement d'images.

4.3 Évaluation de la méthodologie sur l'algorithme de flot optique

L'algorithme de flot optique, présenté dans la section 4.2.1, est utilisé pour évaluer les résultats de la méthodologie de portage sur GPU décrite dans le chapitre 3. Dans un premier temps, nous décrivons, dans la section 4.3.1, les paramètres expérimentaux de cette évaluation. Nous abordons ensuite les résultats des analyses statiques et dynamiques dans la section 4.3.2. Nous évaluons, en section 4.3.3, les critères de placement de la section 3.4. Enfin, en section 4.3.4, nous étudions l'intérêt des transformations de code de la section 3.4.

4.3.1 Protocole expérimental

Dans le cadre de cette évaluation, les mesures sont collectées par instrumentation du code source. Afin de minimiser l'influence de cette instrumentation de code sur le temps d'exécution de l'algorithme, seules les dates des différents évènements sont enregistrées. Les temps d'exécution sont calculés à postériori. Le programme est compilé au moyen de NVCC avec un niveau d'optimisation $-O3$.

L'appel à la fonction principale de cet algorithme est donné dans le listing 4.1. Deux images **img1** et **img2**, de type OpenCV Mat, sont utilisées comme donnée d'entrée pour l'algorithme de flot optique. Celles-ci sont de type High Definition (HD) (1920×1080 pixels) et chaque pixel est codé sur trois composantes couleurs de $8\,bits$. En sortie, le résultat du flot optique est stocké dans l'objet OpenCV Mat dénommé **flow**. Pour les paramètres restants, nous avons utilisé :

- **3** niveaux de sous-échantillonage [1],
- un rayon de **2** pixels pour la fenêtre de recherche du flot optique et
- une distance maximale de flot optique de **4** pixels pour chaque échelle.

Ces paramètres correspondent à plusieurs cas d'application rencontrés pour cet algorithme. Les autres paramètres utilisés sont ceux définis par défaut par l'algorithme. Le jeu d'images en entrée ainsi que les paramètres spécifiés sont conservés à l'identique pour chacun des tests de cette évaluation.

```
calcOpticalFlowSF(img1,img2,flow,3,2,4);
```

Listing 4.1 – Paramètres d'appel de la fonction *simpleflow*

4.3.2 Analyses préliminaires

Nous effectuons, dans un premier temps, les analyses statique et dynamique permettant de construire la représentation spinale de l'algorithme et de déterminer les temps d'exécution des nids de boucles originaux et des différentes fonctions. Le but de ces analyses est d'identifier les nids de boucles naturellement plaçables sur GPU.

Analyse statique – Identification des portions de code portables sur GPU

Notre méthodologie utilise la représentation spinale décrite dans les sections 3.1.6 et 3.1.11, afin de modéliser les caractéristiques de l'algorithme *simpleFlow*. Le résultat de

1. Ce paramètre permet notamment de définir le nombre d'itérations de la boucle l_{26}

cette représentation, générée de manière automatique, est donné en annexe B. Le calcul des dépendances reste pour le moment manuel.

Les résultats mettent en évidence quatre fonctions intégrant un nid de boucles compatible avec les critères de placement sur GPU. Nous retrouvons ainsi dans la représentation spinale les nids de boucles correspondant aux différents appels de fonctions :

— *removeOcclusions* pour $l_{22,23}$, $l_{24,25}$, $l_{105,106}$ et $l_{107,108}$,
— *wd* pour $l_{2,3}$, $l_{12,13}$, $l_{55,56}$, $l_{62,63}$, $l_{73,74}$, $l_{87,88}$ et $l_{109,110}$,
— *calcIrregularityMat* pour $l_{27,28}$ et $l_{41,42}$,
— *calcConfidence* pour $l_{69,70}$ et $l_{83,84}$.

Cependant, une étude attentive de la représentation spinale montre que la fonction *calcConfidence* correspond à une zone de code mort. En effet, les régions exactes pour les tableaux *confidence* et *confidence_inv*, accédées en écriture dans les nids $l_{69,70}$ et $l_{83,84}$, sont totalement modifiées en aval par les accès en écriture des nids $l_{105,106}$ et $l_{107,108}$ de la fonction *removeOcclusions*. De plus, aucun accès en lecture pour l'ensemble des tableaux n'est effectué entre les deux entités. La fonction *calcConfidence* a donc été supprimée de l'algorithme original dans le cadre du portage sur GPU. Le résultat fonctionnel de l'algorithme reste identique après cette suppression.

Le placement initial sur GPU porte donc sur les nids de boucles des trois fonctions restantes : *removeOcclusions*, *wd* et *calcIrregularityMat*.

Modèle de représentation des résultats expérimentaux

Les figures 4.3 à 4.10 représentent les résultats exprimentaux concernant les temps d'exécution sur les plateformes Jetson TX1 (4.1.2) et Endicott (4.1.1). Ce modèle de représentation illustre pour chaque fonction et chaque nid de boucles, leurs dates respectives de démarrage et de fin, relatives à celles de l'application globale. En conséquence, la zone représentée entre ces deux dates correspond au temps d'exécution de l'élément évalué. Les temps d'exécution sont représentés :

— en **gris** pour les **fonctions**,
— en **bleu** pour les **nids de boucles**,
— en **vert** pour les ***kernels*** et
— en **orange** pour les **communications mémoire** hôte/accélérateur.

Les différents temps d'exécution ont été répartis sur six niveaux hiérarchiques, numérotés de 0 à 5. Le premier correspond au cercle le plus interne et le dernier au plus externe. La fonction principale *calcOpticalFlowSF* de l'algorithme *simpleFlow* représente la totalité du niveau 0. Son temps d'exécution correspond à la durée d'exécution globale de l'évaluation. Le temps d'exécution pour chaque fonction et chaque nid de boucles, identifiés au chapitre 3, est systématiquement placé au niveau hiérarchique supérieur dans cette représentation. Le temps d'exécution des *kernels* suit cette même règle.

Le temps évolue dans le sens opposé de celui des aiguilles d'une montre. L'expérimentation débute à la position *'3h'* et se termine à la position *'4h'*. Cette plage fixe correspond au temps d'exécution global de l'algorithme étudié. La taille figée de cette représentation permet :

— de visualiser la répartition des temps d'exécution entre les différents éléments mesurés,
— de mieux identifier les éléments les plus consommateurs en temps d'exécution,
— de favoriser la comparaison entre plusieurs représentations,
— de mieux apprécier les accélérations, pour ces éléments, suite aux transformations appliquées et
— de conserver une vue d'ensemble adaptée au format de ce manuscrit (*A4 portrait*).

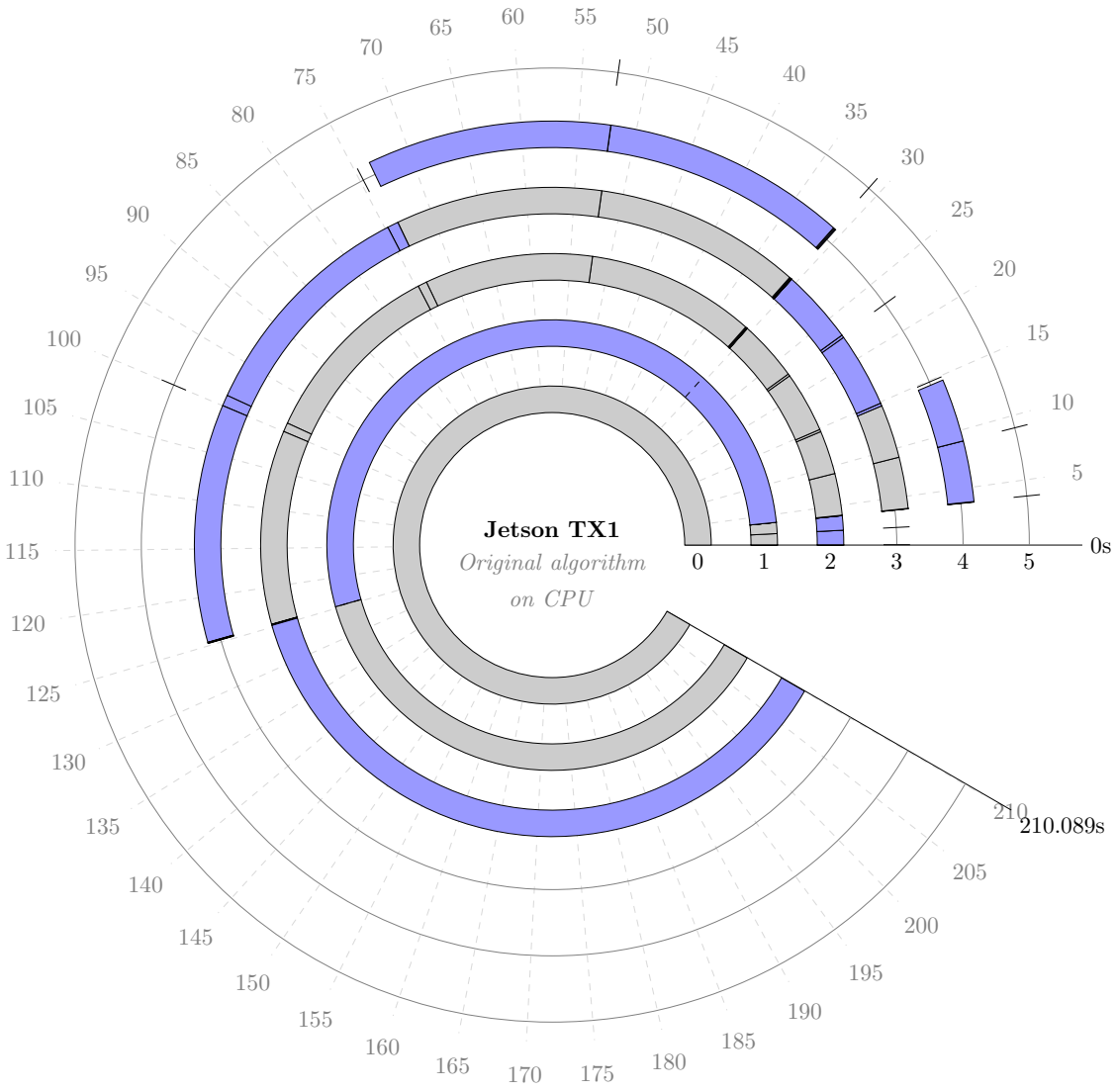

FIGURE 4.3 – Exécution de l'algorithme original *simpleFlow* sur le CPU de la Jetson TX1.

Les éléments les plus consommateurs en temps d'exécution sont plus prioritaires dans le processus de portage afin d'améliorer le temps d'exécution global.

La forme en disques de cette représentation permet d'accentuer les détails des niveaux hiérarchiques les plus élevés, là où la granularité est la plus faible et où les temps d'exécutions ont tendance à être les plus courts. La représentation hiérarchique permet de mieux appréhender les résultats selon une approche inter-procédurale. Enfin, le positionnement des différents temps d'exécution au sein de l'exécution globale de l'application permet de visualiser l'enchaînement des différentes entités exécutées sur l'hôte et sur l'accélérateur.

Pour plus de précision, l'intégralité des données utilisées pour ces représentations est détaillée dans l'annexe D. Les données concernant la plateforme Jetson TX1 sont regroupées dans la section D.1. Celles d'Endicott ont été placées dans la section D.2.

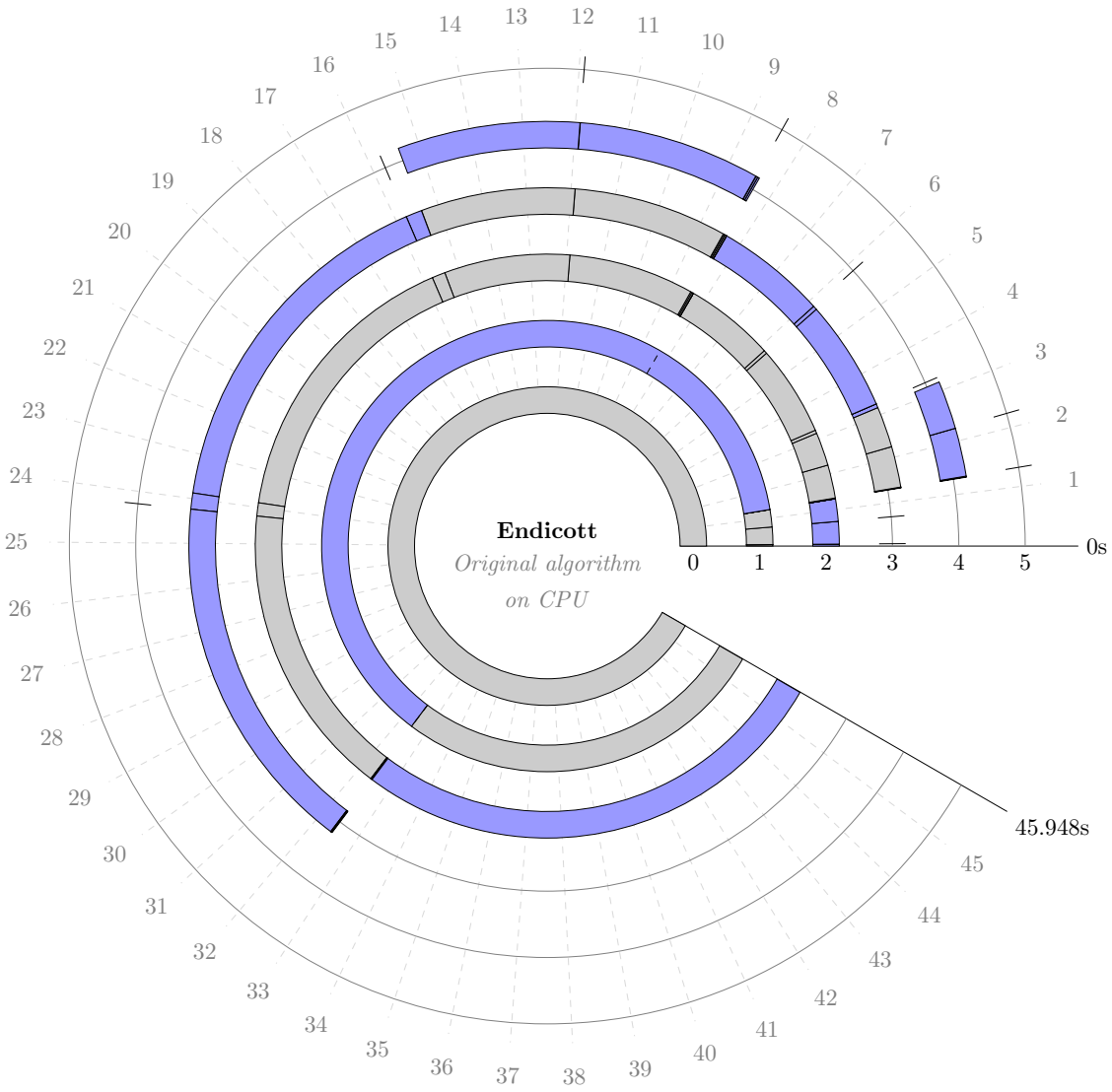

FIGURE 4.4 – Exécution de l'algorithme original *simpleFlow* sur le CPU d'Endicott.

Analyse dynamique – Interprétation de l'exécution de l'algorithme original

L'algorithme original a été écrit pour une exécution séquentielle, correspondant à l'usage d'un unique *thread* placé sur CPU. Celui-ci n'est pas optimisé pour exploiter le maximum de performances d'une architecture CPU donnée. Les temps d'exécution, présentés dans la suite de cette évaluation :

- correspondent donc à l'état original de l'algorithme sans optimisation pour la partie CPU,
- portent exclusivement sur le gain apporté par la méthodologie de portage sur GPU depuis une version non optimisée,
- ne concernent pas les méthodes d'optimisation pour CPU qui sortent du cadre de cette thèse et
- ne permettent pas de déterminer la plateforme la plus adaptée à l'algorithme *simpleflow*.

Les plateformes Endicott et Jetson TX1 présentent pour la partie CPU, une bande

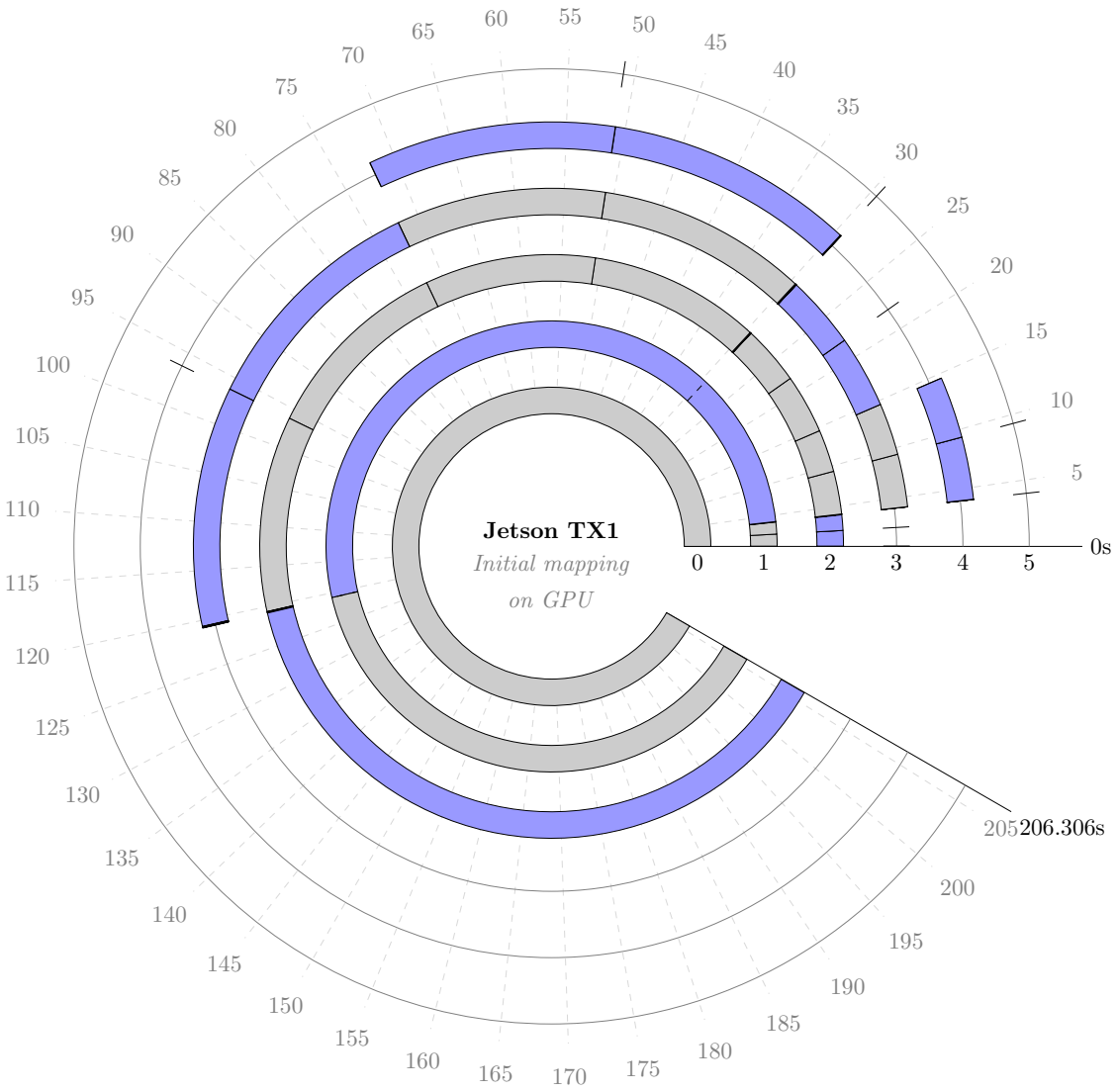

FIGURE 4.5 – Placement initial de l'algorithme Simpleflow sur le GPU de la Jetson TX1

passante mémoire identique de $25.6\,GB.s^{-1}$. Cependant la fréquence de fonctionnement du CPU d'Endicott ($3.9\,Ghz$) est supérieure à celui de la Jetson TX1 ($1.9\,Ghz$). Pour un unique cœur de calcul, le rapport des puissances de calcul simple précision est de 4.1 en faveur d'Endicott. Sans surprise, l'application est exécutée en $45.9\,s$ sur Endicott et en $210.1\,s$ sur la Jetson TX1. Le rapport est alors de 4.57, ce qui reste dans l'ordre de grandeur du rapport théorique exprimé.

Les éléments les plus consommateurs en temps d'exécution sont les fonctions *crossBilateralFilter* et *calcOpticalFlowSingleScaleSF* dont la totalité des appels représente $66,08\%$ et $32,28\%$ du temps global d'exécution sur la Jetson TX1 et 48.49% et 49.05% sur Endicott. La répartition des temps d'exécution n'est donc pas identique entre les deux architectures.

Chaque appel à la fonction *wd* engendre un temps d'exécution extrêment faible ($< 1\,ms$). Son placement sur GPU présente, de ce fait, peu d'intérêt et nous ignorons son portage.

Enfin, la fonction *calcConfidence* représente 1.36% et 1.99% du temps d'exécution global sur la Jetson TX1 et Endicott. Son abandon suppose donc un *speedup* de 1.01 et 1.02.

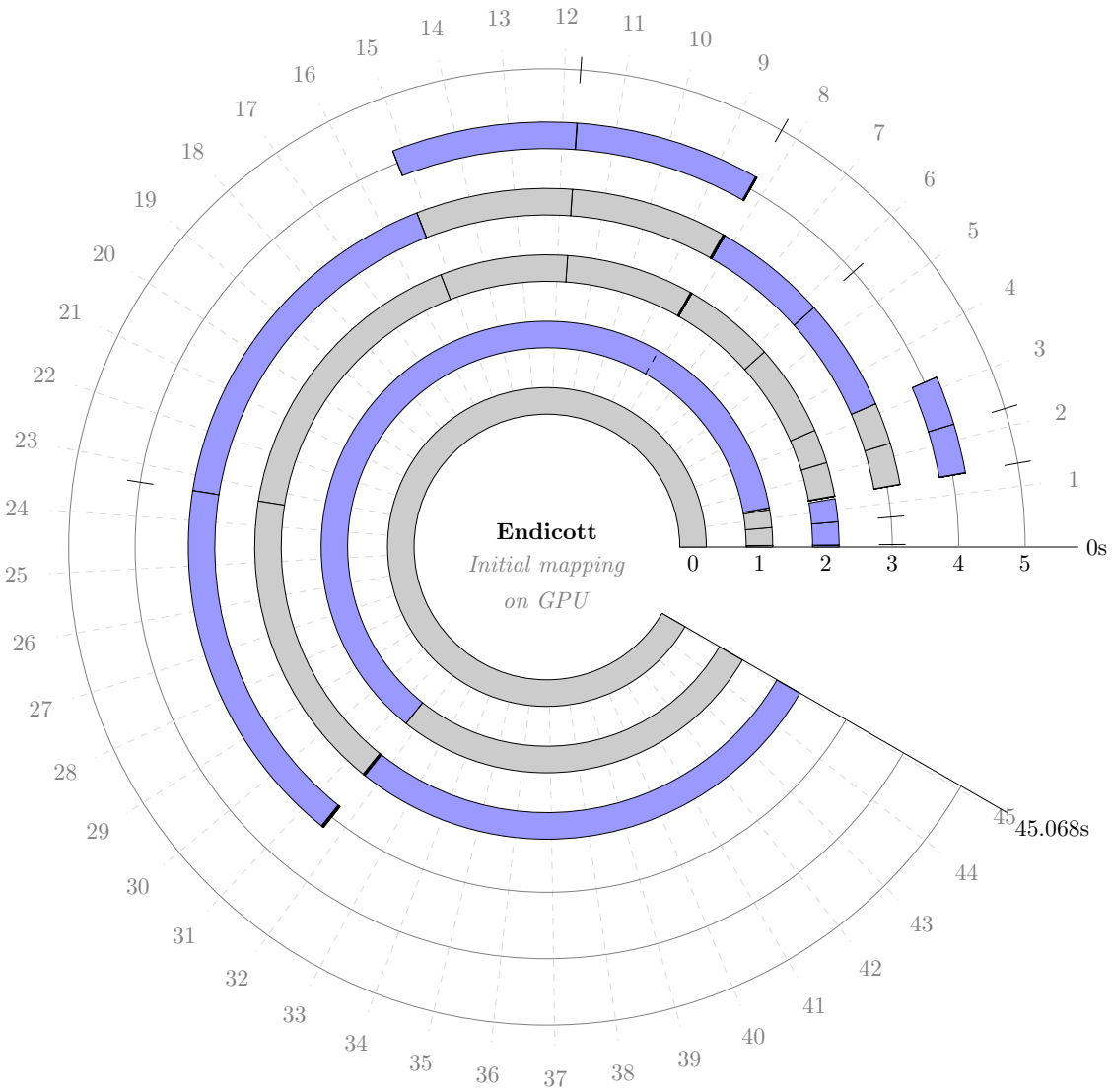

FIGURE 4.6 – Placement initial de l'algorithme Simpleflow sur le GPU d'Endicott

4.3.3 Phase de placement sur GPU

Dans cette partie de l'évaluation, nous utilisons la méthodologie de portage du chapitre 3 sur l'algorithme original. Cependant, nous n'utilisons pas le processus de transformations de code de la section 3.4 qui sera évalué dans la section 4.3.4. Seul le *tiling* de boucles est utilisé pour répartir les itérations des boucles parallèles sur les instances de *blocks* et de *threads* du GPU. Cette approche se rapproche ainsi de la vectorisation de boucles, classiquement utilisé pour le portage sur GPU [16].

Suite à l'analyse préliminaire de la section 4.3.2, seuls deux nids de boucles issus des fonctions *removeOcclusions* et *calcIrregularityMat*, sont portés sur GPU. Les *kernels removeOcclusions_kernel* (listing C.5) et *calcIrregularityMat_kernel* (listing C.1) sont alors générés ainsi que leurs communications mémoires CPU/GPU. Conformément à l'algorithme original, *removeOcclusions_kernel* est exécuté six fois dans la globalité de l'algorithme et *calcIrregularityMat_kernel* quatre fois.

Pour rappel, les fonctions *wd* et *calcConfidence* ont été écartées du processus de portage.

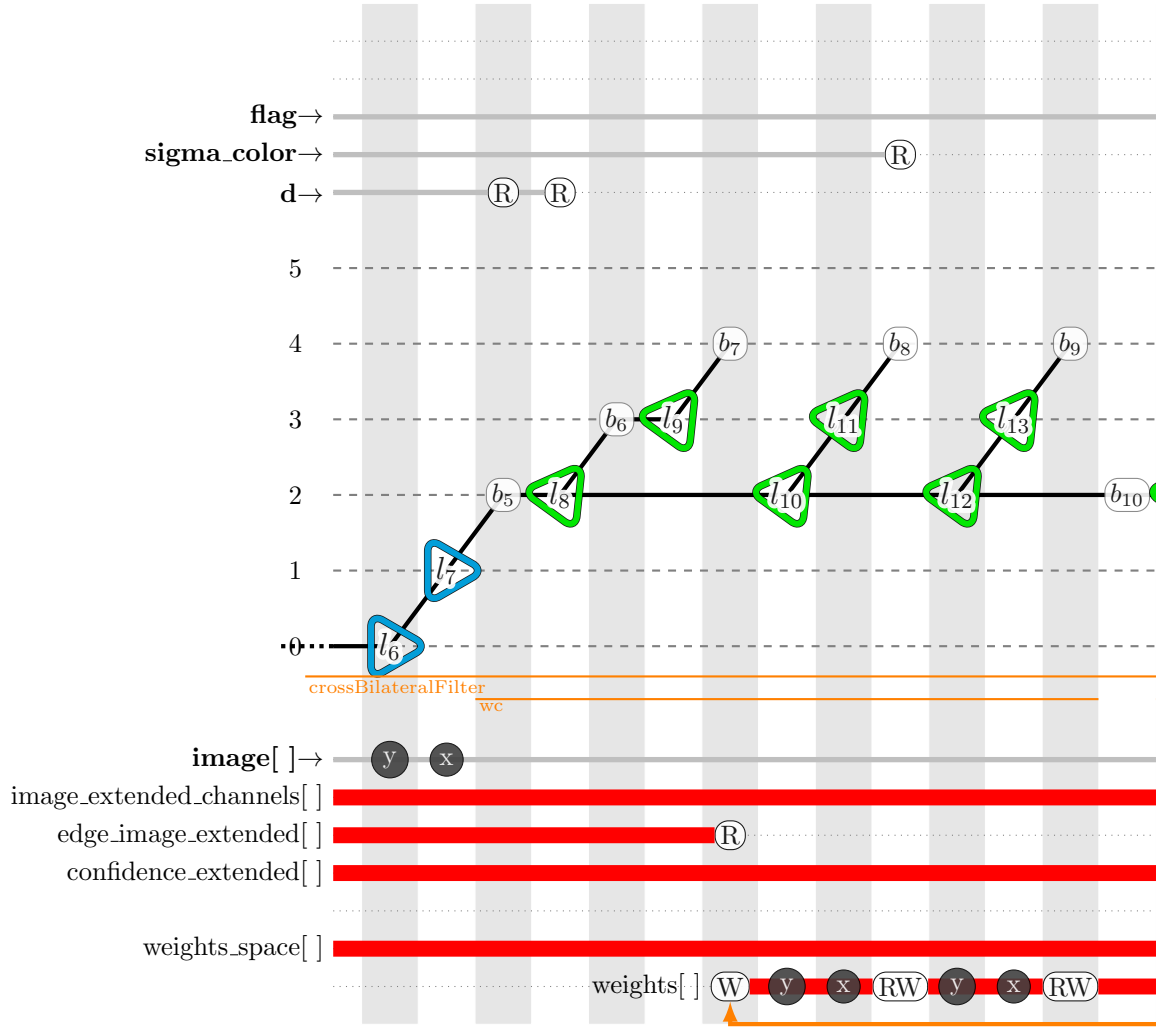

FIGURE 4.7 – Extrait de représentation spinale pour la fonction *crossBilateralFilter* (1/2)

Le temps d'exécution global visible dans la figure 4.5 passe ainsi de 210.1 *s* à 206.3 *s* sur la Jetson TX1. Pour Endicott, le temps d'exécution de la figure 4.6 passe de 45.9 *s* à 45.06 *s*. Le *speedup* dans les deux cas est de 1.02.

Dans la mesure où les deux fonctions concernées par le portage sur GPU représentent une part extrêment faible du temps d'exécution global, nous constatons sans surprise que le *speedup* reste proche de 1.

La fonction ***removeOcclusions*** subit globalement une dégradation de son temps d'exécution, suite à son placement sur GPU. Cette dégradation est imputable aux communications mémoire CPU/GPU. Le *speedup* moyen sur TX1 est de 0.3 et celui sur Endicott de 0.2. La pénalité globale de ce placement correspond à 0.125 *s* sur TX1 et 0.096 *s* sur Endicott

En revanche, ***calcIrregularityMat*** bénéficie globalement d'une accélération suite à son placement sur GPU. Sur TX1, le *speedup* moyen est de 3.9 et sur Endicott, celui-ci est de 4.1. Le gain de ce placement est de 0.185 *s* sur TX1 et 0.06 *s* sur Endicott

Nous remarquons tout de même que la majorité du *speedup* global, pour l'application, est lié à l'abandon de *calcConfidence*. L'ensemble des exécutions pour cette fonction représente dans le cadre de l'algorithme initial, 2.868*s* sur TX1 et 0.914*s* sur Endicott. Cela correspond respectivement à 76% et 103.8% de l'accélération du temps d'exécution global

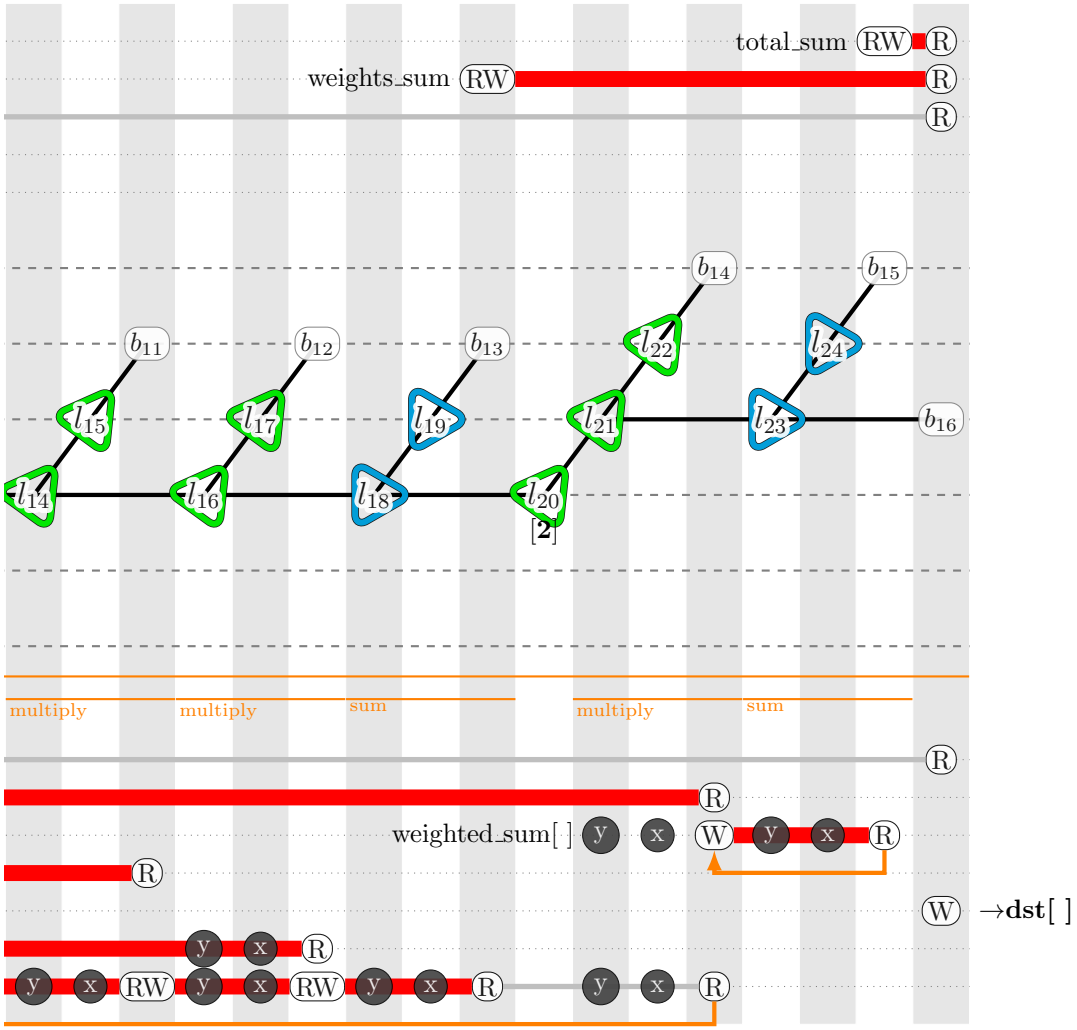

FIGURE 4.7 – Extrait de représentation spinale pour la fonction *crossBilateralFilter* (2/2)

obtenu, en considérant le placement pénalisant de *removeOcclusions*.

Suite à cette première phase de placement, nous concluons :

— qu'aucun nid de boucles, parmi ceux ayant les temps d'exécution les plus importants, n'est plaçable sur GPU et
— que l'utilisation exclusive de critères de placement, tels que ceux développés dans la section 3.3, ne permettent pas de considérer suffisamment de nids de boucles pour engendrer une accélération significative.

Ce constat n'est pas imputable à l'imbrication parfaite ou non des nids de boucles et aux espaces d'itération normés des nids de boucles concernés. Nos critères de placement ne considèrent pas ces paramètres. La normalisation des espaces d'itérations de la section 3.5.3 ainsi que le déplacement de blocs inter-boucles GPU de la section 3.5.2 assurent les critères du placement.

Les transformations de code de la section 3.4 sont donc nécessaires, afin d'améliorer la quantité de placement du GPU.

4.3.4 Amélioration de la quantité de code placé sur GPU

Nous appliquons à présent le processus de transformations de code introduit dans la section 3.4.

À l'issu de ce processus, plusieurs nids de boucles deviennent compatibles avec les critères de placement :

— $l_{[4,11]}$, $l_{[14,21]}$, $l_{[75,82]}$ et $l_{[89,96]}$ pour les différents appels à la fonction *calcOpticalFlowSingleScaleSF* et

— $l_{[57,61]}$, $l_{[64,72]}$ et $l_{[111,115]}$ pour la fonction *crossBilateralFilter*.

crossBilateralFilter

La fonction *crossBilateralFilter* a déjà été utilisée, pour illustrer les transformations de code, dans la section 3.4. L'extrait de sa représentation spinale est repris dans la figure 4.7. Afin de simplifier les explications, nous désignons les différents éléments, sujets aux transformations de code, dans le référentiel de cette représentation. Ainsi, l'ensemble des transformations du nid de boucles $l_{[6,24]}$ sont applicables aux nids de boucles $l_{[57,61]}$, $l_{[64,72]}$ et $l_{[111,115]}$ de la représentation spinale globale, située en annexe B.

Les boucles $l_{6,7}$ présentent une dépendance embarquée, symbolisée par une flèche générant un cycle, sur les tableaux *weights* et *weighted_sum*. Cette dépendance empêche, selon le critère 1 (section 3.3.1), le placement de ces boucles sur les instances de *block* du GPU et donc la création d'un *kernel* à ce niveau. L'analyse des dépendances indique que l'intégralité des données, pour ces deux tableaux, présentent une antidépendance [2], entre les itérations de ces deux boucles. Afin de rendre les boucles $l_{6,7}$ parallèles, il est possible d'appliquer au niveau du corps de la boucle l_7 :

— une expansion ou

— une privatisation de ces tableaux.

En s'appuyant sur les analyses statique et dynamique, nous vérifions la validité de ces solutions. Les tableaux *weights* et *weighted_sum* sont alloués de manière dynamique. Leur taille est similaire et dépend du paramètre d. Celle-ci correspond à :

$$|M_{weights}| = |M_{weighted\_sum}| = (2 \times d + 1)^2 * 4\,Bytes$$

L'analyse dynamique nous donne $d = 18$ pour cette évaluation. L'analyse statique nous permet de retrouver, dans la représentation spinale globale, le lien entre d et les paramètres de l'application :

— *upscale_averaging_radius* pour $l_{[57,61]}$ et $l_{[64,72]}$ et

— *postprocess_window* pour $l_{[111,115]}$

Ces paramètres ont pour valeur par défaut 18 mais sont modifiables par l'utilisateur.

La privatisation de *weights* implique alors l'allocation en *heap memory* de $5.35\,KB$ de données pour chacune des itérations des boucles $l_{6,7}$. La privatisation de *weighted_sum* engendre la même quantité de données allouée.

L'analyse dynamique donne pour les différents appels à la fonction *calcOpticalFlowSingleScaleSF* :

— 1080, 540 et 270 itérations pour la boucle l_6 et

— 1920, 960 et 480 itérations pour la boucle l_7.

Ces valeurs correspondent à la taille des images d'entrées (1920×1080 pixels) ainsi que leurs différents niveaux de sous-échantillonage selon un rapport de décimation par deux pour chaque dimension. Le nombre de niveaux d'échantillonage dépend du paramètre *layers*

2. symbolisée par la couleur orange de la flèche

défini lors de l'appel à la fonction principale. Celui-ci a été fixé à 3 pour cette évaluation (listing 4.1).

La quantité globale de données, allouées pour le nid de boucles $l_{[6,24]}$, augmenterait au maximum de $2 \times 5.35\,KB \times 1920 \times 1080 = 21.15\,GB$ pour cette évaluation. Le critère 3 (section 3.3.3), dans ce cas, ne serait plus respecté pour les deux plateformes utilisées. En conséquence, l'expansion de tableau ne permet pas de répondre au critère 3 de placement.

À la différence de l'expansion de tableau, la quantité de mémoire requise par la privatisation de tableau dépend du nombre de *threads* exécutés simultanément sur le GPU. Par défaut, $8\,MB$ peuvent être alloués, au maximum, en *heap memory* par *kernel*. La table 4.2 indique l'exécution possible de 2048 *threads* par SM pour les deux plateformes. Ces dernières disposent chacune de deux unités SM, ce qui représente 4096 *threads* exécutés simultanément. Cela représente $4096 \times 5.35KB = 42.8\,MB$ de données utilisées en *heap memory*. Cette valeur est supérieure à la limite [3] par défaut du GPU.

L'allocation dynamique de mémoire étant coûteuse, nous proposons, parmi l'ensemble des solutions de placement possible, une privatisation de ces tableaux à un niveau de profondeur plus élevé dans le nid de boucles. Cette solution correspond au listing C.3 dans l'annexe C. Les boucles $l_{8,9}$, $l_{10,11}$, $l_{12,13}$, $l_{14,15}$, $l_{16,17}$, $l_{18,19}$, $l_{21,22}$ et $l_{23,24}$ ont le même nombre d'itérations. Ces boucles sont parallèles à l'exception de $l_{18,19}$ et $l_{23,24}$ qui présentent un cas de réduction.

La boucle l_{20} ayant un nombre connu d'itérations (deux itérations), nous procédons à l'**unrolling** complet de celle-ci, afin de ramener $l_{21,22}$ et $l_{23,24}$ au même niveau que les autres boucles. L'analyse des dépendances nous permet d'effectuer la **fusion** de ces boucles dans les boucles $l_{8,9}$.

Nous procédons enfin à la privatisation, dans le corps de la boucle l_9, des tableaux :
- *weights* sous la forme de la variable scalaire *weight* et
- *weighted_sum* qui n'utilise plus qu'un registre temporaire lors de la réduction des données dans le scalaire *total_sum*.

Suite à ces deux privatisations :
- il n'y a plus de tableau alloué dynamiquement dans la mémoire de l'accélérateur,
- les boucles $l_{6,7}$ deviennent parallèles et
- les trois critères de placement sont satisfaits et permettent le placement de $l_{6,7}$ sur les instances de *blocks* du GPU.

Cependant, suite à la *fusion*, les boucles $l_{8,9}$ deviennent séquentielles.

Dans le cas où $d < 15$, il est possible selon le critère 2 (section 3.3.2) de placer les boucles $l_{8,9}$ sur les instances de *threads* du GPU, car :

$$\forall d \in \mathbb{N},\ (2 \times d + 1)^2 \leq 1024 \Rightarrow d \leq 15$$

Il est dans ce cas nécessaire d'employer une **réduction parallèle** ou une instruction *atomicAdd*, afin de préserver les dépendances embarquées sur les scalaires *total_sum* et *weights_sum*.

Cependant, dans le cadre de cette évaluation, nous sommes dans le cas $d > 15$. Nous avons choisi, afin d'obtenir une solution de placement plus générique, de répartir les itérations des boucles $l_{6,7}$ sur les instances de *blocks* et de *threads* du GPU, en appliquant un **tiling**.

Suite à son placement sur GPU, la fonction *crossBilateralFilter* connait une accélération moyenne de 58.47 sur TX1 (figure 4.9) et 8.77 sur Endicott (figure 4.10). Ce gain nous permet de justifier l'intérêt des transformations de code dans notre méthodologie.

3. Cette limite peut être étendue au moyen du paramètre *cudaLimitMallocHeapSize*

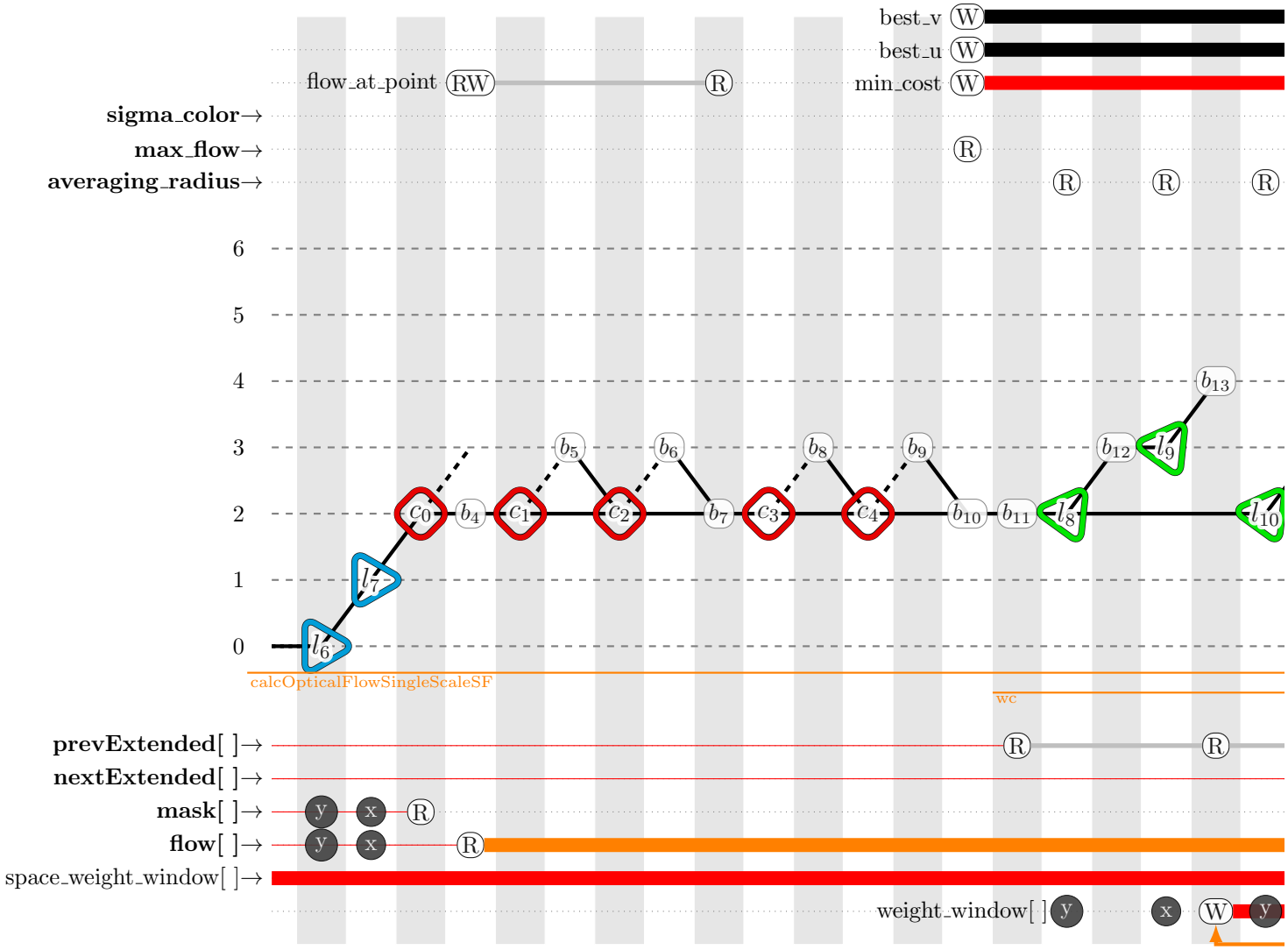

FIGURE 4.8 – Extrait de représentation spinale pour la fonction *calcOpticalFlowSinglsScaleSF* (1/2)

calcOpticalFlowSingleScaleSF

Nous étudions à présent la fonction *calcOpticalFlowSingleScaleSF*. L'extrait de sa représentation spinale est visible dans la figure 4.8. Comme pour la fonction *crossBilateralFilter*, nous nous référons aux différents éléments selon le référentiel de cette dernière. Les différentes transformations effectuées sur le nid de boucles $l_{[6,19]}$ sont transposables aux boucles $l_{4,11}$, $l_{14,21}$, $l_{75,82}$ et $l_{89,96}$ de la représentation spinale globale, située dans l'annexe B.

Nous remarquons que :

— les boucles $l_{6,7}$ présentent une antidépendance sur le tableau *weight_window*,
— les boucles $l_{16,17}$ présentent :
  — une dépendance de flot sur le scalaire *min_cost*,
  — une dépendance de sortie sur le scalaire *best_u* et
  — une dépendance de sortie sur le scalaire *best_v*,
— les boucles $l_{18,19}$ présentent une dépendance de flot sur le scalaire *cost*,
— les boucles $l_{8,9}$, $l_{10,11}$, $l_{12,13}$,$l_{14,15}$ sont parallèles.

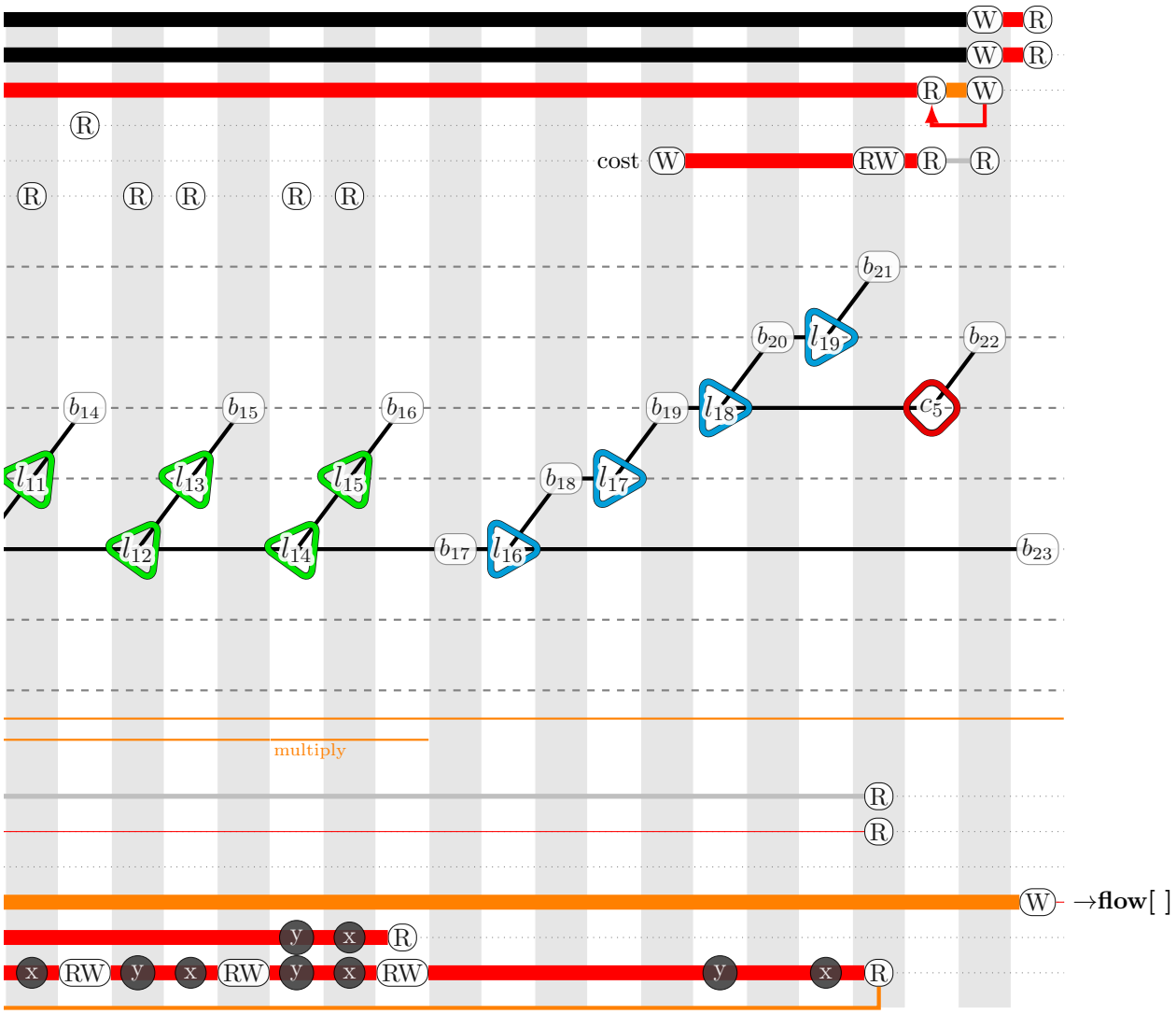

FIGURE 4.8 – Extrait de représentation spinale pour la fonction *calcOpticalFlowSinglsScaleSF* (2/2)

L'analyse dynamique donne pour les différents appels à la fonction *calcOpticalFlowSingleScaleSF* :

— 1080, 540 et 270 itérations pour la boucle l_6 et
— 1920, 960 et 480 itérations pour la boucle l_7.

Comme pour la fonction *crossBilateralFilter*, ces valeurs correspondent à la taille des images d'entrée (1920×1080 pixels) ainsi que leurs différents niveaux de sous-échantillonage. Le nombre d'itérations pour ces deux boucles dépend des données en entrée et ne peut être déduit par une simple analyse statique. Enfin, le branchement conditionnel c_0 dépendant des données du tableau *mask*, nous considérons ces domaines d'itération selon une enveloppe dense et convexe pour chaque niveau.

Les boucles $l_{18,19}$, ainsi que $l_{8,9}$, $l_{10,11}$, $l_{12,13}$,$l_{14,15}$, dépendent du paramètre *averaging_radius* de l'application. Le nombre d'itérations pour ces boucles correspond à $2 \times averaging\_radius + 1$. Ce paramètre a été affecté à 2 dans le cadre de cette expérimentation (listing 4.1), ce qui donne 5 itérations pour chacune de ces boucles.

Enfin, les boucles l_{16} et l_{17} dépendent indirectement du paramètre *max_flow*. Le

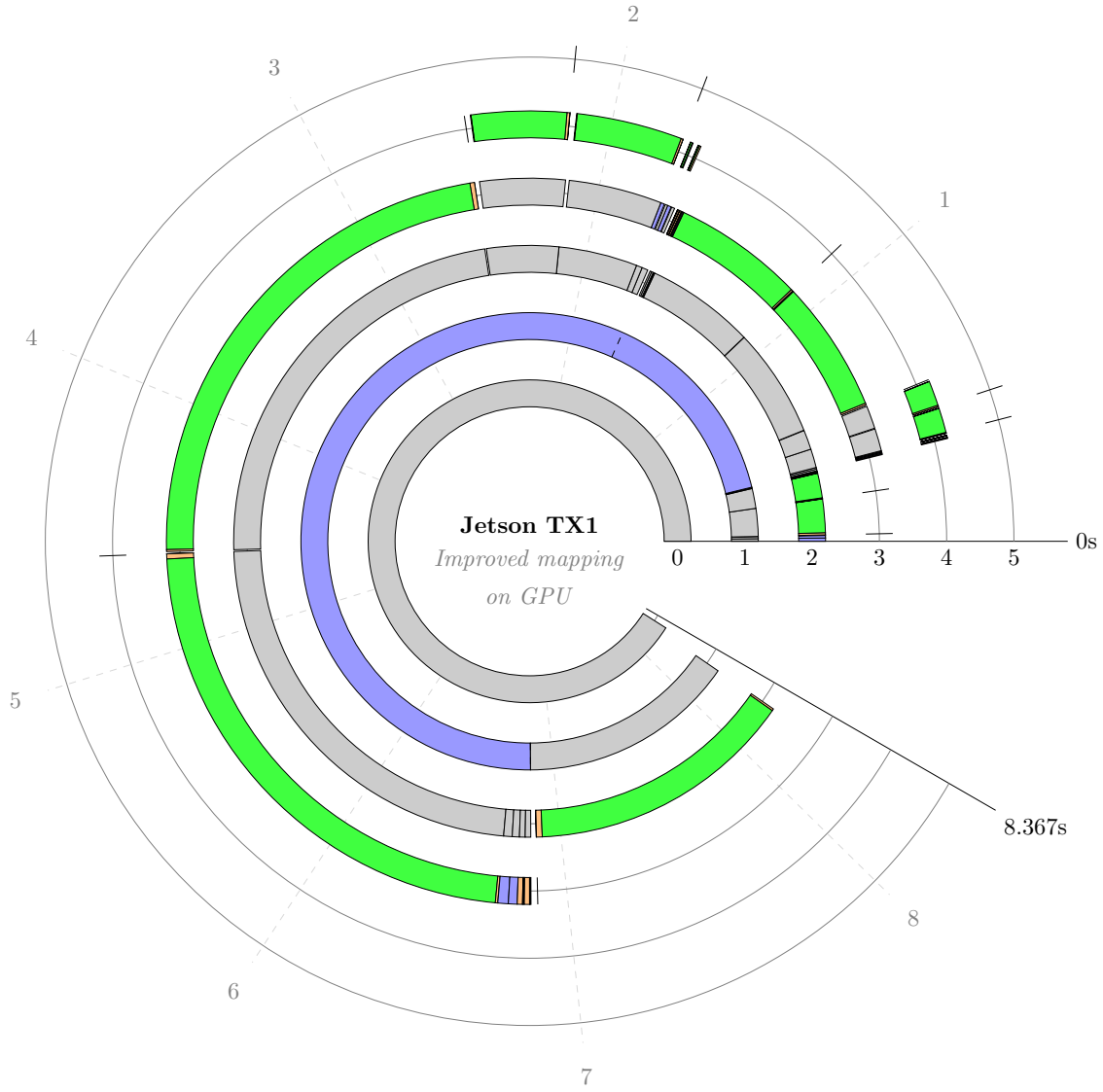

FIGURE 4.9 – Amélioration de la quantité de placement sur le GPU de la Jetson TX1

nombre d'itérations est bien plus complexe à définir. Nous utiliserons de ce fait une enveloppe convexe d'itération correspondant à $2 \times max\_flow + 1$. Ce paramètre a été défini avec une valeur de 4 dans le cadre de cette expérimentation (listing 4.1), ce qui donne 9 itérations pour chacune des deux boucles.

Les bornes pour l'ensemble des boucles de la fonction *calcOpticalFlowSingleScaleSF* sont donc dynamiques et ne peuvent être déterminées par la simple utilisation des analyses statiques.

Notre solution de placement, correspondant au listing C.2 dans l'annexe C, considère les transformations de code suivantes. La problématique est similaire à celle de la fonction *crossBilateralFilter* pour les boucles $l_{6,7}$. Le critère 1 (section 3.3.1) de placement n'est pas respecté, les boucles $l_{6,7}$ étant séquentielles. L'expansion de tableau ne permet pas de repecter le critère 3 (section 3.3.3) de placement. La privatisation, sous forme d'un scalaire, du tableau *weight_window* permet :

— de lever la dépendance embarquée pour ces deux boucles,
— de placer les itérations de ces deux boucles sur les instances de *blocks* du GPU.

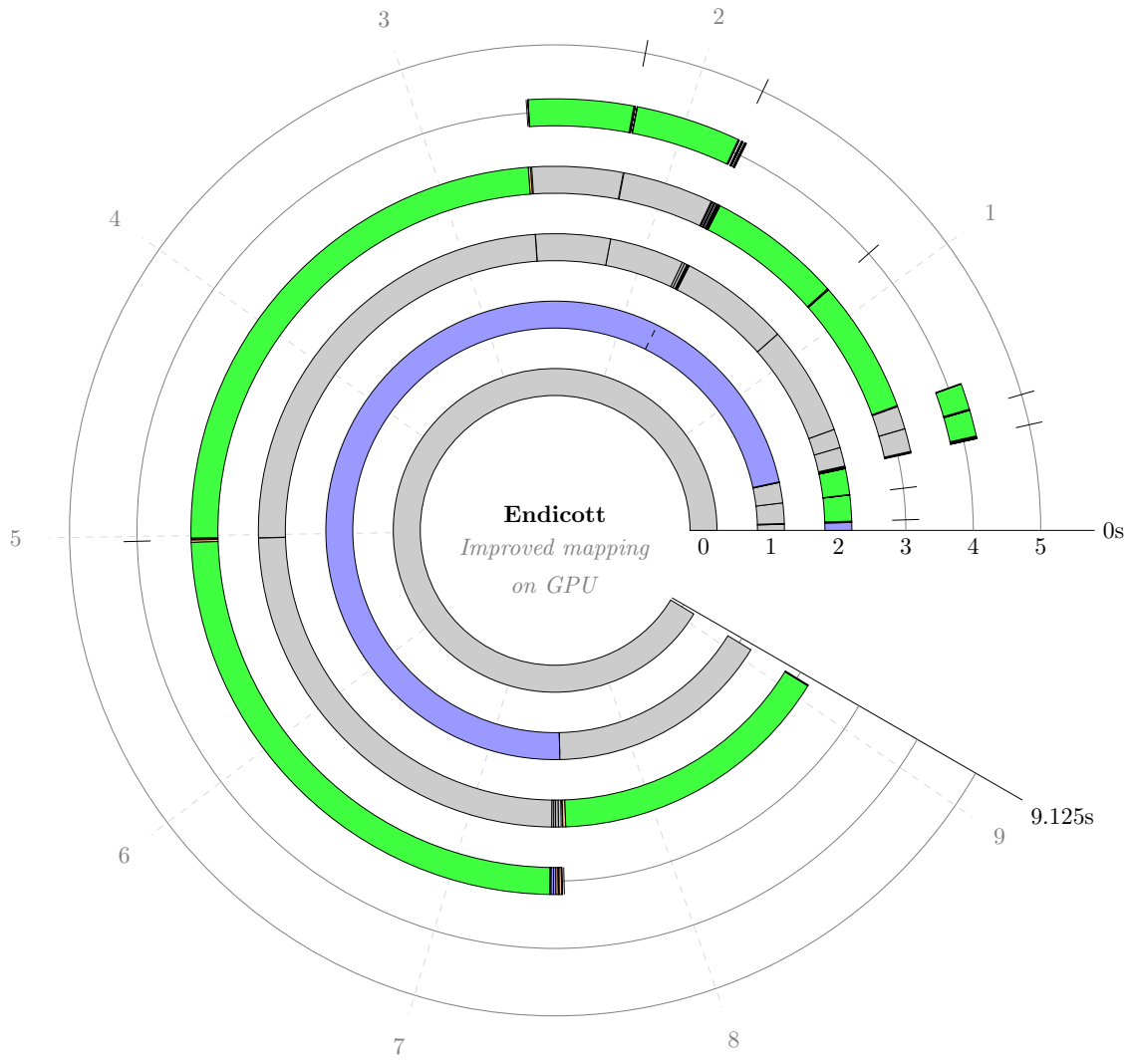

FIGURE 4.10 – Amélioration de la quantité de placement sur le GPU d'Endicott

La **fusion** des boucles $l_{10,11}$, $l_{12,13}$,$l_{14,15}$ dans la boucle $l_{8,9}$ ne pose pas de problème particulier du fait du typage parallèle de ces boucles. Cette transformation améliore au passage la localité temporelle des données. Cependant, le niveau de profondeur des boucles $l_{18,19}$ diffère de celui des boucles $l_{8,9}$. Après analyse des dépendances, nous fusionnons l'ensemble de ces boucles dans les boucles $l_{18,19}$. Cette solution a pour conséquence d'augmenter le nombre d'opérations et de communications mémoire des boucles $l_{[8,15]}$ par un facteur correspondant au nombre d'itérations des boucles $l_{16,17}$. En revanche, suite à cette transformation :

- le tableau *weight_window*, n'est plus alloué dynamiquement dans la mémoire de l'accélérateur,
- le tableau *weight_window* est privatisé sous forme la forme de la variable scalaire *weight* dans la boucle l_19,
- les boucles $l_{6,7}$ deviennent parallèles et
- les trois critères de placement sont satisfaits et permettent le placement de $l_{6,7}$ sur les instances de *blocks* du GPU.

Enfin, nous appliquons un ***tiling*** sur les boucles $l_{6,7}$. En conséquence de l'augmentation du nombre d'opérations et de communications mémoires, la pression sur les registres

augmente. La taille des tuiles est alors réduite, chaque *block* étant composé de 640 *threads*, afin que le nombre d'unités registre utilisées restent dans les limites des caractéristiques du GPU (table 4.2).

Cette solution nous permet de bénéficier d'une accélération moyenne de 12.56 sur TX1 et de 3.96 sur Endicott. Le gain est ici plus limité que celui de la fonction *crossBilateralFilter*. Cela s'explique par l'augmentation du nombre d'opérations et de communications mémoire ainsi que la taille réduite du *tiling*, réduisant le nombre de *threads* par *block*.

Conclusion sur les transformations de code

Après les transformations de code, nous obtenons un *speedup* global sensiblement plus important, avec 24.66 sur TX1 et 4.94 sur Endicott. La différence s'explique par l'écart entre les puissances de calcul des CPUs de la TX1 et d'Endicott. Le gain significatif obtenu provient de l'importance de ces deux fonctions dans le temps d'exécution global.

Point intéressant, suite aux transformations, le temps global d'exécution de l'application est similaire pour les deux plateformes. Cependant, la tendance s'inverse et la TX1 devient plus rapide ($8.367\,s$) qu'Endicott ($9.125\,s$). L'architecture basse consommation de la TX1 nous apporte, au final, les meilleures performances en temps d'exécution pour cette évaluation.

En conclusion, notre processus de transformations permet d'améliorer la quantité de code placé sur GPU. Cela se traduit par une amélioration significative du temps d'exécution global. De plus, dans le cadre de l'architecture embarquée basse consommation du TX1, notre évaluation met en évidence la nécessité du placement de code sur la partie GPU du SOC T210, afin de sensiblement réduire le temps d'exécution cette architecture.

4.3.5 Conclusion sur l'évaluation de la méthodologie

Nous avons montré, avec cette évaluation, l'importance de chacune des phases de la méthodologie dans le processus de placement sur GPU et en particulier leurs impacts sur les temps d'exécution.

La phase d'**analyse statique** nous a servi, dans la section 4.3.2 :

- à constituer la représentation spinale de l'algorithme étudié, regroupant l'ensemble des informations nécessaires au placement sur GPU et
- à identifier le code mort des opérations effectuées par la fonction *calcConfidence* dans l'algorithme *simpleflow*.

La phase d'**analyse dynamique** nous a permis, dans la section 4.3.2 :

- d'évaluer les nids de boucles les plus consommateurs en temps d'exécution et ainsi de prioriser leur portage sur GPU,
- d'identifier les fonctions, telle que *wd*, ayant un temps d'exécution trop faible ($< 1\,ms$) pour un placement sur GPU et
- de reconnaître les fonctions, telle que *removeOcclusions*, dont les performances, suite à leur portage sur GPU, sont dégradées par les échanges mémoire hôte/accélérateur.

Les **critères de placement**, développés dans la section 3.3, ont permis d'obtenir un placement fonctionnel sur GPU, dans les sections 4.3.3 et 4.3.4. Cependant, la simple identification des nids de boucles compatibles avec l'architecture GPU, n'était pas suffisante pour obtenir une accélération significative du temps d'exécution global.

Le processus de **transformations de code**, décrit dans la section 3.4, a apporté (dans la section 4.3.4) une augmentation de la quantité de nids de boucles adaptés aux critères de placement. En conséquence, le temps d'exécution global a bénéficié, grâce à ce

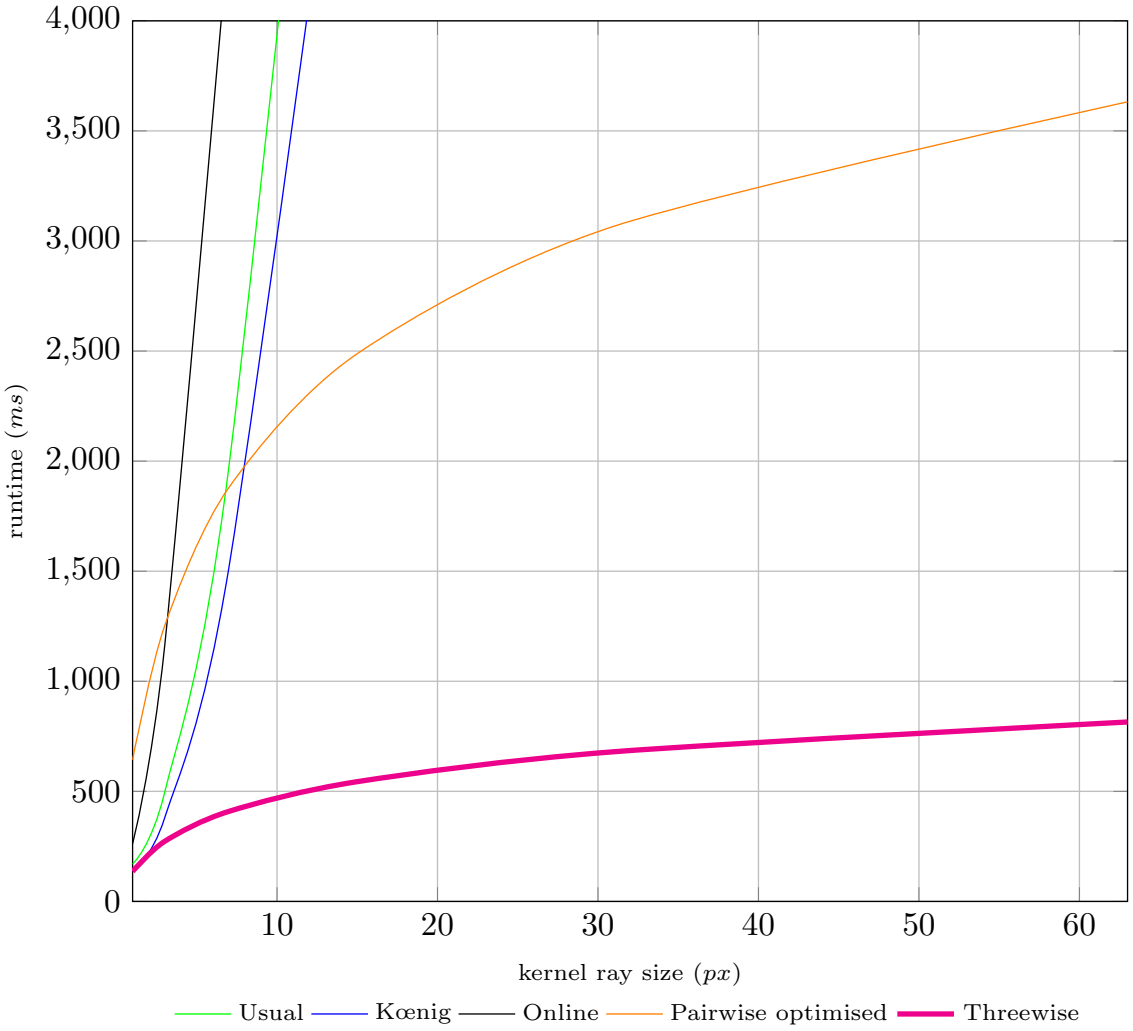

FIGURE 4.11 – Temps d'exécution de l'algorithme de variance locale en fonction de la taille du voisinage

processus, d'une accélération significative de 24.66× et 4.94× respectivement sur les deux architectures évaluées.

Enfin, nous avons constaté que le CPU ARM de la Jetson TX1 dispose d'une puissance de calcul plus limitée. Notre méthodologie de placement sur GPU a apporté, pour cette plateforme basse consommation, un gain sensible de 24.66× sur le temps d'exécution global. Nous en déduisons que le placement sur GPU est indispensable pour le SOC *T210*, afin de réduire le temps d'exécution.

4.4 Évaluation des transformations de code sur l'algorithme de variance locale

Dans le cadre de notre méthodologie, nous avons défini un ensemble de transformations (section 3.4) permettant d'améliorer la quantité et la qualité des placements sur GPU. Dans cette section, nous abordons spécifiquement le cas des réductions parallèles. Deux publications [63, 64] présentent nos travaux et nous les résumons dans cette section.